UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Generador Automático de Personajes 3D

Raed Ahissami Yordi
Tutor: Carlos Aguirre Maeso

Julio 2017

Generador automático de personajes 3D

AUTOR: Raed Ahissami Yordi
TUTOR: Carlos Aguirre Maeso

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2017

Resumen (castellano)

Este proyecto, denominado GenPro, consiste en un generador procedural de modelos humanoides 3D. Con esta herramienta podremos generar un número incalculable de modelos distintos a partir de un único modelo. El generador esta implementado sobre la plataforma de desarrollo de videojuegos Unity3D como un *asset* programado en C#. GenPro puede ser exportado e importado por cualquier usuario de la plataforma.

Actualmente, la generación procedural es empleada en juegos de alta gama para reducir, en la medida de lo posible, el espacio ocupado por el videojuego, el tráfico de internet necesario para la obtención del modelo o el tiempo de lectura del modelo desde el soporte físico. La generación procedural es quizás el método de programación de videojuegos más importante hoy en día. Aunque parezca que es un método nuevo de programación, la generación por procedimientos viene usándose desde casi el principio de la creación de los videojuegos.

El proyecto consta de dos partes. La primera es la edición del personaje, donde podremos elegir el género, la altura, los rasgos faciales o el color de piel entre otras características. Además, si lo deseas podrás generarlo de manera aleatoria. Una vez finalizada la edición del modelo a nuestro gusto podremos entrar con nuestro personaje en un escenario lleno de diversos personajes generados aleatoriamente por todo el mapa. Nuestro personaje podrá moverse por todo el terreno buscando personajes y viendo todas las diferencias físicas entre cada uno de ellos.

Palabras clave (castellano)

GenPro, generador procedural, Unity3D, asset, modelos humanoides

Abstract (English)

This project, called GenPro, is composed of a procedural generator of 3D humanoid models. With this tool we can generate an untold number of models from a single model. The generator is implemented on the Unity3D video game development platform as an active programmed in C #. GenPro can be exported and imported by any platform user.

Currently the procedural generation is used in high-end games, in order to reduce, as far as possible, the space occupied by the video game, the internet traffic necessary to obtain the model or the reading time of the model from the physical medium. The procedural generation is perhaps the most important method of video game programming today. Although it seems to be a new method of programming, the generation by procedures has been used since the beginning of the creation of video games.

The project has two parts. The first is the edition of the character, where you can choose the gender, height, facial features or skin color among other features. Also, if you want you can generate it randomly. Once finished editing the model to our liking we will be able to enter with our character in a scene full of diverse personages generated randomly by all the map. Our character will be able to move throughout the terrain looking for characters and seeing all the physical differences between each one of them.

Keywords (inglés)

GenPro, procedural generator, Unity3D, asset, humanoid models

Agradecimientos

Me gustaría agradecer a todas las personas que me han ayudado y apoyado a lo largo de la carrera y especialmente durante el desarrollo de este trabajo:

En primer lugar, me gustaría agradecer a Carlos por haberme dado la oportunidad de desarrollar este proyecto.

A todos los compañeros y amigos que he hecho a lo largo de estos años en la universidad. Especialmente a Isa, Edu y Antonio, por tantas horas de risas y agobios compartidas.

A mis amigos de toda la vida, los del barrio y el instituto, que, aunque no entendiesen el significado de ‘la práctica no me compila, hoy no puedo salir’, han creído en mí y me han ayudado a seguir adelante.

A mi familia, por absolutamente todo, por su apoyo y amor incondicional, por darme la fortaleza y un empujón para salir adelante cuando lo he necesitado.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

1 Introducción

Este Trabajo de Fin de Grado tiene como propósito desarrollar un generador automático de modelos humanoides en 3D. En este apartado se hablará de los motivos que me han llevado a la realización de este proyecto, de sus objetivos y de la estructura de la presente memoria.

1.1 Motivación

La generación procedural/procedimental, también llamada generación automática o algorítmica, es la creación o generación de contenido pseudoaleatorio en base a unas reglas y conductas determinadas. Esto quiere decir que el contenido generado no está creado de antemano si no que, a partir de uno o varios modelos, se pueden generar prácticamente un número ilimitado de contenidos.

La eficacia de la programación procedimental no radica en la libertad que nos da, si no en la manera en la que nos la ofrece. Siempre va a haber unas pautas básicas y dentro de las cuales existe la aleatoriedad. Si no existiesen unas medidas impuestas por el programador y existiese la libertad absoluta podría perjudicar la jugabilidad o crear contenido sin sentido (modelos, vehículos, ciudades). Por ejemplo, en nuestro caso, si estamos generando personajes humanos de manera procedural tendríamos que tener en cuenta una serie de parámetros y valores (tamaño del brazo, tamaño del tronco, piernas, etc.) para que no nos quede un modelo anómalo o deforme.

La generación procedural se ha popularizado en estos últimos años, pero no es algo novedoso ya que empezó a utilizarse décadas atrás. Sin embargo, actualmente la tecnología permite que este método de programación vaya mucho más allá de lo imaginable. Lo distinguido de este método, y su fundamental ventaja, reside en su capacidad para reducir en la medida de lo posible el espacio ocupado por el videojuego, el tráfico de internet necesario para la obtención del modelo o el tiempo de lectura del modelo desde el soporte físico. Además, su implementación ahorra tiempo de desarrollo en el futuro del proyecto.

Como dice *Oliver Franklin-Wallis* en su artículo *Games of the future will be developed by algorithms, not humans* en el futuro las superproducciones de videojuegos cambiarán su tipo de desarrollo a este método procedimental, ya que para ellos será mucho más rentable (en costes de desarrollo, de cantidad de programadores y de tiempo). Por tanto, los mundos no serán creados por los desarrolladores, si no por los algoritmos. Por otra parte, la utilización de la generación procedural permitirá el diseño de videojuegos más grandes y más variados.

No solo existe la programación procedural en videojuegos, también puede llevarse a cabo en muchos otros campos, como por ejemplo la música o el cine.

Este Trabajo de Fin de Grado ha sido realizado con la intención de que otros usuarios de *Unity* puedan utilizarlo y añadir a sus proyectos un generador automático de personajes añadiendo ellos mismos el propio modelo a utilizar o usando el dado.

1.2 Objetivos

Este proyecto se centra en el desarrollo de un generador automático de modelos 3D creado como un *asset* para Unity. En este Trabajo de Fin de Grado se alcanzan dos objetivos principales. El primer objetivo es la creación de una amplia variedad de modelos humanoides a partir de un único modelo, donde cada uno de ellos tenga características bien diferenciadas. El segundo es poder utilizar este asset en cualquier proyecto de Unity donde se quiera usar la edición de personajes/modelos o la generación de muchos modelos.

A continuación se describirán las fases que se han seguido para cumplir con los objetivos citados:

1. Implementación de un *script* para la edición de un modelo humano. A partir de un modelo cualquiera (modelo humanoide, con brazos, piernas, etc.) poder editarlo arbitrariamente. Para ello en nuestro proyecto dispondremos de una escena inicial donde estará nuestro prototipo con el *script* añadido y una serie de barras de edición para modificar los parámetros del personaje.

2. Desarrollo de un *script* que dote a nuestro personaje creado de una animación o controlador que deseemos.

3. Diseño de un *script* para una segunda escena donde se generará el número que queramos de modelos distintos a lo largo de un mapa, colocados en posiciones aleatorias y con distintas animaciones.

4. Realización de pruebas en dos escenas auxiliares

    a. En la primera utilizaremos un modelo distinto al principal para comprobar que el *script* creado funciona con distintos modelos (Cambiando el nombre de los huesos del modelo en el *script* o remplazando los del modelo por los que se usan en el *script*).

    b. La segunda escena de pruebas sirve para comprobar que no generamos personajes extraños y que siempre sale un modelo bien construido. Generamos 10 personajes de manera procedural y al pulsar un botón se reemplazan por otros 10 modelos.

1.3 Organización de la memoria

La memoria está dividida en 5 capítulos además de esta introducción. Los capítulos son los siguientes:

- **Estado del arte:** En el capítulo dos se habla sobre el estado del arte actual en relación al proyecto realizado y las aplicaciones y los usos relacionados con la generación procedural.
- **Diseño:** En el capítulo tres definiremos el proyecto, cuál es su alcance y cuáles son sus funcionalidades. También se detallarán los requisitos de los diferentes módulos que componen el proyecto. Además, hablaremos de los entornos de programación elegidos y cuál ha sido el motivo de su utilización. Para terminar se detallará el modelo de ciclo de vida del trabajo que se ha llevado a cabo.
- **Desarrollo:** En el cuarto capítulo se hablará sobre la implementación del proyecto.
- **Pruebas y resultados:** En este capítulo se exponen las pruebas realizadas para verificar el correcto funcionamiento del *asset*, el tipo de pruebas ejecutadas y los resultados obtenidos.
- **Conclusiones y trabajo futuro:** En este último capítulo se exponen las conclusiones sobre el trabajo realizado y las futuras mejoras y modificaciones a realizar.

2 Estado del arte

En este capítulo se muestra el estado y el estudio de las tecnologías y aplicaciones actuales relacionadas con este proyecto. También hablaremos sobre las ventajas y los inconvenientes de la utilización de un generador procedural.

2.1 La generación procedural

En estos últimos años la generación por procedimientos de contenido parece ser la última novedad en generación de mapas de videojuegos, generación de enemigos, texturas, música o efectos especiales, pero lo cierto es que no es una técnica reciente y lleva siendo un método utilizado desde el inicio de la creación de los videojuegos para los primeros ordenadores.

Como hemos mencionado antes, una de las ventajas de utilizar la generación procedural (PCG) es que no requiere un excesivo almacenamiento en el disco duro. Por eso a principios de los 80 apareció la generación procedural en los videojuegos, ya que estos, debían adaptarse a los ordenadores y a las limitaciones que presentaban. El primero de ellos fue *Elite,* lanzado en 1984, que contenía 8 galaxias y más de 200 planetas. Todo ello en ordenadores de 8 bits. La manera en la que se conseguía generar cada uno de los planetas con sus características sin que se guardara en el disco duro era con la PCG. Se utilizaban una semilla numérica con unos valores determinados que al pasarla por el algoritmo generaba la composición de los planetas. De esta manera cada vez que se iniciaba el juego, los datos de cada galaxia y planeta se cargaban sin ser almacenados.

La generación procedural fue pasando poco a poco a un segundo plano a medida que los ordenadores iban evolucionando y teniendo mejores características técnicas. Una de las razones es que, al diseñar cada nivel o cada personaje de manera manual, éste tiene un nivel de detalle mayor que de manera algorítmica, además de que requiere mayor velocidad en el procesamiento de datos.

En la última década, con la mejora de las herramientas utilizadas para la generación procedural y con el fin de competir contra las grandes compañías por parte de las empresas pequeñas, se volvió a popularizar este método. Uno de sus mayores objetivos es el de crear contenido muy distinto y muy amplio en un mismo juego.

Actualmente la generación procedural no es algo únicamente aplicado en el mundo de los videojuegos, sino que también es utilizada en otros campos como el cine o la música. Por ejemplo, en el cine tenemos un paquete software llamado *MASSIVE* desarrollado por Stephen Regelous que fue creado para la generación de efectos visuales de *El Señor de los Anillos,* para generar automáticamente una batalla entre ejércitos de miles de soldados. Básicamente *MASSIVE* es un simulador de multitudes donde a cada objeto o persona creada le asigna un movimiento o una IA, y entre ellos interactúan como lo haría un grupo de personas reales. A partir del éxito que tuvo, *MASSIVE* evolucionó y fue utilizado por otras muchas producciones en televisión. También se llevó a otros campos, siendo

utilizado para la arquitectura, la ingeniería civil, para la planificación peatonal, el transporte, para la seguridad y muchos campos más.

3 Diseño

En este capítulo se puntualiza de forma más detalla el Trabajo de Fin de Grado y su alcance. Además, se explicarán los requisitos del proyecto, así como su estructura y las herramientas que se van a utilizar para implementarlo. Finalmente hablaremos del ciclo de vida que seguiremos.

3.1 Definición del proyecto

El proyecto consistirá en crear un generador procedural de modelos 3D. El generador se implementará sobre la plataforma de desarrollo de videojuegos *Unity3D* como un *asset*. Se pretende realizar un programa que pueda crear, a partir de un único modelo, un número incalculable de modelos distintos. A estos modelos creados a partir de un algoritmo se les podrá añadir cualquier tipo de animación o controlador deseado. Adicionalmente, también se ha añadido un editor para la modificación del modelo a nuestro gusto.

3.2 Alcance

El proyecto consta de dos partes, la primera seria la edición del modelo, donde podremos modificar todas las partes que queramos del personaje. Así, podremos observar los rangos y las pautas que hemos definido para el generador automático (¿Como de grande puede llegar a ser?, ¿Cuál es la altura mínima que puede tener?, ¿Que formas puede tener la nariz?, etc.). Y la segunda parte se centra en el propio generador de modelos, el cual dota a dichos modelos de una animación concreta como añadido a su función.

Se aspira a que cualquier usuario de la plataforma, que quiera utilizar un editor de personajes y/o tener un generador automático de modelos humanoides, se pueda descargar el *asset*, ya que podrá ser exportado e importado por cualquiera.

Como hemos comentado anteriormente en otros apartados, una de las principales ventajas de un generador procedural es reducir en la medida de lo posible el espacio ocupado por el videojuego. Si quisiéramos tener una amplia variedad de personajes secundarios o enemigos lo único que necesitaríamos es un modelo humanoide y este generador procedural.

3.3 Análisis de requisitos

En este apartado definiremos los requisitos funcionales y los no funcionales de nuestro proyecto, tanto de la edición del modelo como del generador procedural del mismo modelo.

3.3.1 Requisitos funcionales

Editor de modelos

RF-EM 1. El usuario podrá elegir el género del personaje. El modelo tendrá una achura diferente, un tamaño de pechos distinto y otra textura dependiendo del genero escogido.

RF-EM 2. El usuario dispondrá de un botón para poder generar su modelo de manera aleatoria.

RF-EM 3. El usuario podrá seleccionar la edición de la cara, donde podrá editar los siguientes atributos:

- Tamaño y forma de los ojos
- Tamaño del tabique nasal
- Tamaño de las aletas de la nariz
- Tamaño y curvatura general de la nariz
- Volumen de las mejillas
- Grosor de los labios
- Longitud de los labios
- Tamaño del mentón
- Anchura de la quijada
- Tamaño de las orejas
- Tamaño de la cabeza

RF-EM 4. El usuario podrá seleccionar la edición del cuerpo, donde podrá editar los siguientes atributos:

- Color de piel
- Tamaño del cuello
- Tamaño de los hombros
- Tamaño del tronco
- Anchura de la cintura
- Longitud de los brazos
- Anchura de los brazos
- Tamaño del pectoral
- Volumen del cuerpo en general
- Volumen del abdomen
- Anchura de las caderas
- Anchura de las piernas
- Longitud de las piernas

RF-EM 5. El usuario tendrá la opción de empezar la edición del modelo desde el principio.

RF-EM 6. El usuario podrá finalizar la edición del modelo y pasar a la siguiente escena.

RF-EM 7. El usuario podrá cerrar la escena y terminar con la edición del modelo.

Generador procedural de modelos

RF-GM 1. El usuario podrá crear de manera algorítmica el número de modelos distintos que desee.

RF-GM 2. El usuario podrá añadirles animación o un controlador a los modelos.

RF-GM 3. En la escena creada una vez finalizada la de edición, el usuario podrá pasearse por el mapa y visitar los distintos modelos originados.

3.3.2 Requisitos no funcionales

RNF 1. El *asset* implementado deberá poder ser utilizado por cualquier usuario de la plataforma *Unity*.

RNF 2. Todas las opciones de edición y los botones que tienen las escenas deberán estar bien distribuidas a lo largo y ancho de la pantalla, ajustándose a cualquier tamaño.

RNF 3. Todas las operaciones realizadas deberán responder de forma inmediata.

3.4 Estructura de huesos del Modelo 3D

En este apartado podemos ver la estructura anatómica que tiene nuestro modelo y cada uno de los huesos que lo compone. Estos huesos son sobre los que el algoritmo actuará.

3.4.1 Estructura del cuerpo del modelo

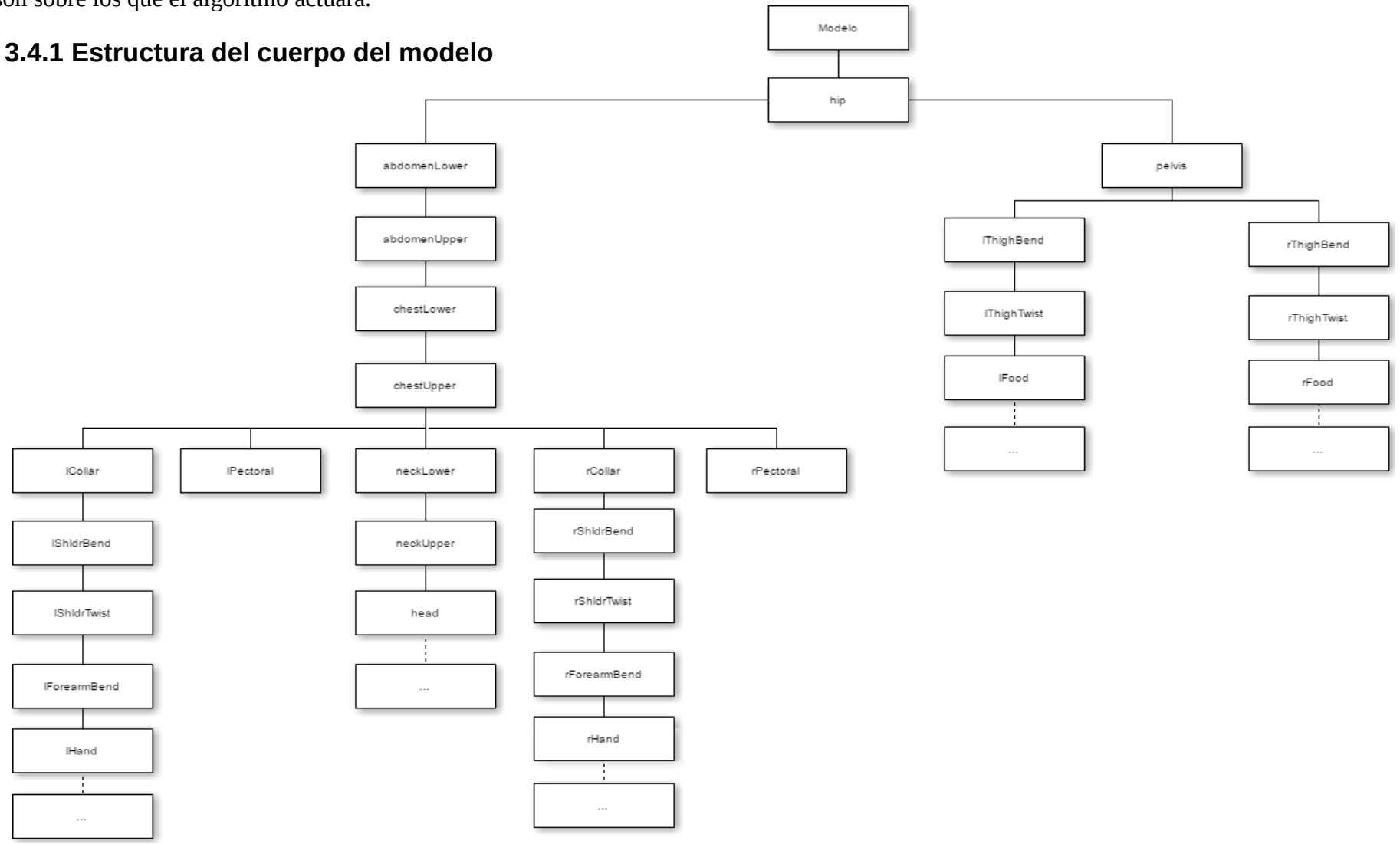

Figura 3.4.1 Huesos del cuerpo del modelo

3.4.2 Estructura de la cabeza del modelo

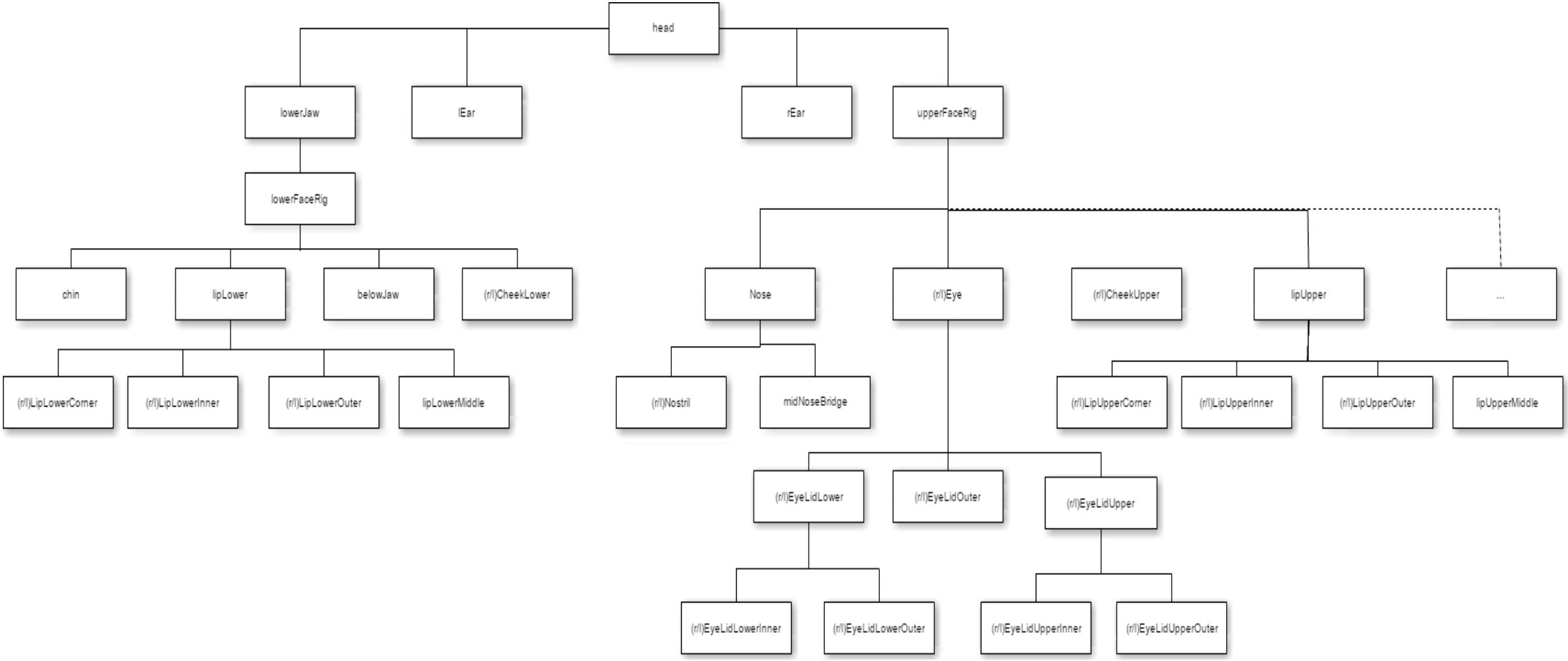

Figura 3.4.2 Huesos de la cara del modelo

3.5 Tecnología y herramientas utilizadas

En este apartado hablaremos de las tecnologías de desarrollo y de las herramientas que se van a utilizar para la implementación del proyecto.

3.5.1 Tecnologías

Los *scripts* creados para el proyecto GenPro, que podrían incluirse en cualquier *asset* de *Unity*, se han implementado en C#. Se ha elegido este lenguaje de programación en vez de javascript porque, además de ser más potente y utilizado para grandes proyectos de videojuegos, la mayoría de los *assets* en *Unity* están programados en C# (Aunque en un mismo proyecto puedas tener *scripts* programados en ambos lenguajes).

3.5.2 Herramientas

A continuación, detallaremos las herramientas a utilizar para desarrollar el presente proyecto.

- *Unity*: Es una de las herramientas para desarrollar videojuegos más completas que existen actualmente. Una de las razones para ser tan utilizada es porque permite la creación multiplataforma desde un único desarrollo (podemos programar para Android y para iOS a la vez).

- *Visual Studio*: Ha sido el entorno de desarrollo utilizado para la creación de todos los scripts del proyecto.

- *Bitbucket*: Es un sistema de control de versiones basado en *Git* utilizado para guardar el estado del proyecto en cada instante deseado. De esta manera, si una nueva funcionalidad añadida a nuestro proyecto sale mal o estropea otras funcionalidades, podemos volver a una versión anterior donde todo funcionaba.

- *Cacoo:* Es un servicio web diseñado para crear diagramas. Los diagramas y esquemas que se han creado para este documento han sido hechos en *Cacoo*.

3.6 Modelo de ciclo de vida

Este proyecto sigue un modelo de ciclo de vida incremental e iterativo dividido en 4 etapas:

- Análisis
- Diseño
- Implementación
- Pruebas

Gracias a este modelo de ciclo de vida y junto con *Bitbucket*, podemos tener una versión estable y funcional del proyecto desde el desarrollo del primer incremento. Con este modelo de ciclo de vida se pueden añadir nuevos requisitos si se presentan.

Los incrementos serán aproximadamente los siguientes:

Iteración 1: Creación de una escena en *Unity* para la edición de un modelo. Selección del género y edición de los rasgos faciales.

Iteración 2: Edición del cuerpo del modelo. Y creación de un botón que genere el modelo con unos rangos aleatorios.

Iteración 3: Se realizarán los siguientes requisitos funcionales: RF-EM 5, RF-EM 6 y RF-EM 7

Iteración 4: Se cumplirán los requisitos funcionales RF-GM 1, 2 y 3.

Iteración 5: El proyecto cumplirá todos los requisitos funcionales y no funcionales.

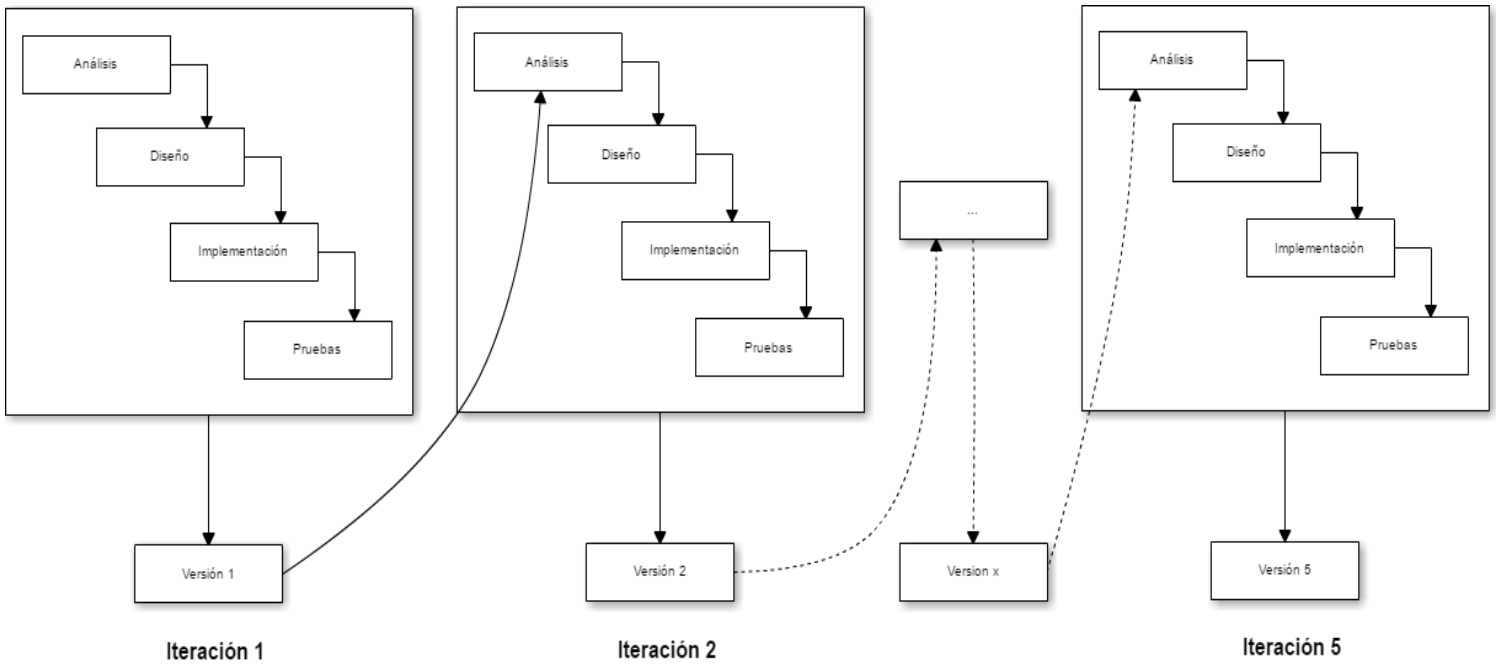

Figura 3.6 Ciclo de vida del proyecto

4 Desarrollo

En este capítulo se explicará el desarrollo del *asset* descrito en los apartados anteriores.

4.1 Estructura del proyecto

La estructura del asset creado es la siguiente:

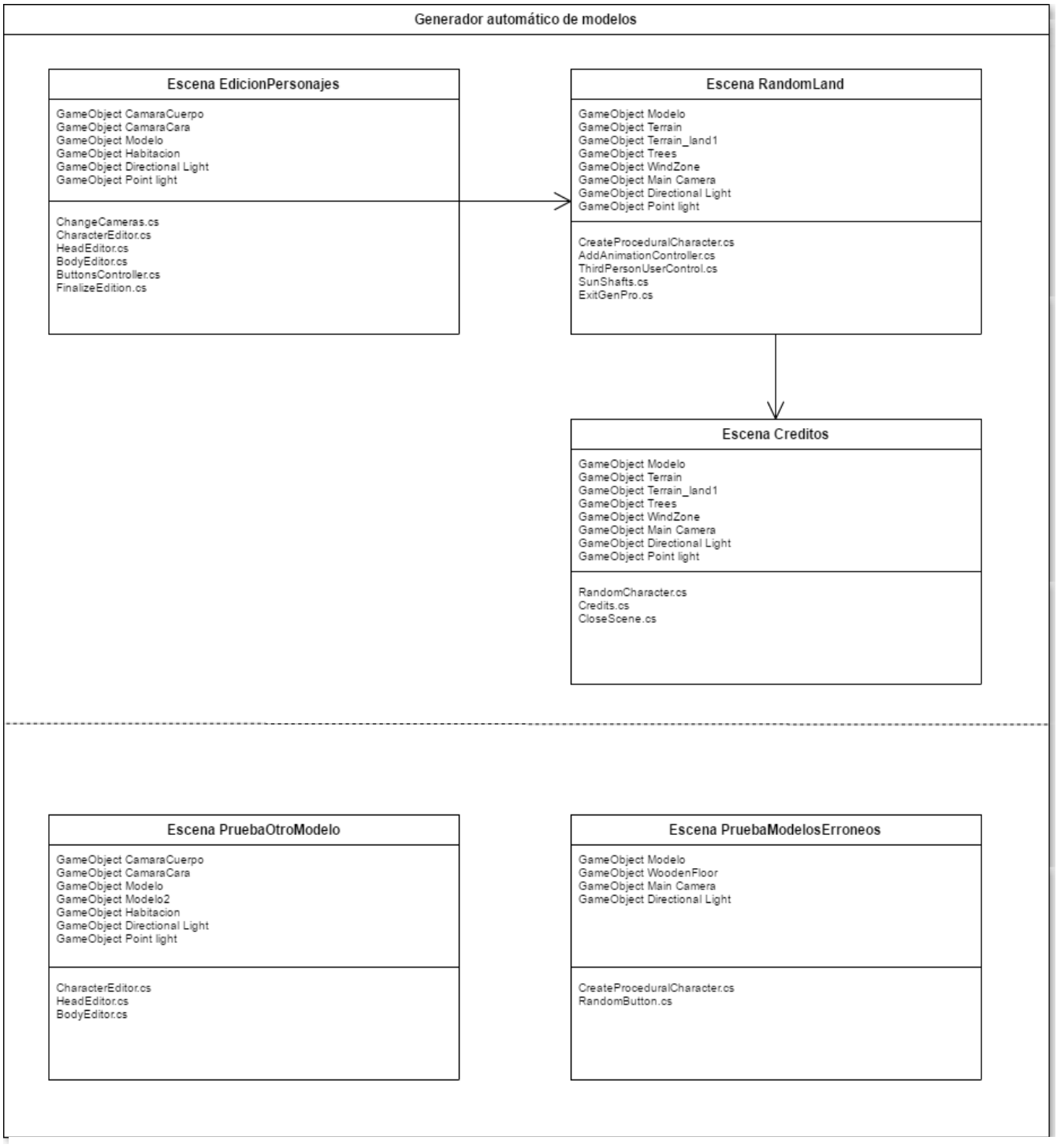

Figura 4.1 Estructura del asset

Cada uno de los cuadrados representa una escena del *asset*, con sus *GameObjects* y los *scripts* necesarios para el correcto funcionamiento.

4.2 Composición del Asset

En este apartado hablaremos de cada una de las escenas que componen nuestro *asset*, de los *GameObjects* que la forman y de cada uno de los *scripts* que contienen.

4.2.1 Escena EdicionPersonajes

Esta escena es la del editor del modelo, donde podremos modificar todas las partes que queramos del personaje.

Los *GameObjects* que componen esta escena son los siguientes:

- **CamaraCuerpo:** Es la cámara principal, está activa cuando empieza la escena. Si pulsamos en el botón de edición de cara se desactiva.

- **CamaraCara:** Esta cámara apunta a la cabeza del modelo. Se activa si pulsamos la edición de cara y se desactiva al pulsar edición de cuerpo.

- **Modelo:** Es el modelo humanoide que queremos editar. Este *GameObject* está compuesto por cada uno de los huesos del modelo, que son los hijos de este. Estos *GameObjects* hijos son a los que accederemos con el *script*.

- **Habitacion:** Es la habitación donde se encontrará nuestro modelo al ser editado.

- **Directional light:** Determina la iluminación de la habitación y el sombreado deseado de los GameObjects que componen la escena.

Los Scripts que se utilizan en esta escena son los siguientes:

- **ButtonsController.cs:** Crea todos los botones que hay en la escena (*'Hombre', 'Mujer', 'Aleatorio'*, '*Editar Cara'*, *'Editar Cuerpo', 'Empezar de nuevo' y Finalizar*) y llama al script correspondiente al pulsar cada botón.

- **ChangeCamara.cs:** Aquí tenemos los métodos necesarios para los botones '*Editar Cara'* y *'Editar Cuerpo'.* Cuando pulsamos en ellos activa y desactiva la cámara correspondiente.

- **FinishEdition.cs:** Este *script* lo utilizamos para cambiar de escena. Aquí tenemos los métodos necesarios para el botón *'Finalizar',* al darle clic cambiaremos a la siguiente escena.

- **CharacterEditor.cs:** Aquí creamos los métodos para la funcionalidad de los botones '*Hombre'*, '*Mujer'*, '*Aleatorio'* y '*Empezar de nuevo'*. Los dos primeros botones modifican el modelo y le cambian la textura según el

botón seleccionado. Si pulsamos '*Aleatorio'* nos generara un modelo con unos rangos aleatorios (siempre dentro de unos rangos determinados, evitando que el modelo quede deforme). Al pulsar en '*Volver a empezar'* el modelo volverá a los parámetros originales.

También este *script* nos crea *horizontal sliders* para cada una de las partes del cuerpo a editar (ojos, anchura de la nariz, tamaño de la boca, longitud de las piernas, etc.).

En el siguiente código puede verse uno de los métodos más utilizados en este *script*. Este método nos permite acceder a cualquier hueso del modelo. Introduciendo el nombre del modelo y el nombre del hueso obtenemos el *GameObject* al que pertenece el hueso y podremos modificarlo a nuestro gusto.

```
static public GameObject obtenerHuesoModelo(GameObject modelo, string hueso){

    Transform[] bones = modelo.transform.GetComponentsInChildren<Transform>(true);
    foreach (Transform bone in bones) if (bone.gameObject.name == hueso){
            return bone.gameObject;
        }
    return null;
}
```

Figura 4.2.1.1: Método ObtenerHuesoModelo

- **HeadEditor.cs:** Contiene los métodos para modificar los rasgos faciales del modelo. Estos métodos serán llamados en CharacterEditor.cs.

- **BodyEditor.cs:** Contiene los métodos para modificar el cuerpo del modelo. Estos métodos serán llamados en CharacterEditor.cs.

  Con este otro ejemplo de código podemos ver como creamos el *slider* para el cambio de color de piel y como lo cambiamos según los parámetros del *slider*.

```
/*Color de piel*/
GUI.Label(new Rect(Screen.height, 25, 100, 20), "Color de piel");
pielColor = GUI.HorizontalSlider(new Rect(Screen.height, 45, 100, 40), pielColor, 1f, 0.1f); /*Creamos el slider para el cambio de color de piel*/
Debug.Log("Color de piel " + pielColor);

if (GUI.changed)
{
    /*Asignamos los valores del slider a la variable piel*/
    piel.a = 1f;
    piel.r = pielColor;
    piel.g = pielColor;
    piel.b = pielColor;

    cambiarColorPiel(Modelo, piel); /*asignamos el color al modelo*/
}
```

Figura 4.2.1.2: Código para cambiar el color de piel

A continuación, podemos observar cómo está estructurado la escena en Unity, a la izquierda podemos observar los GameObjects, en el centro la pantalla de edición de la escena y a la derecha como se vería al ejecutarlo.

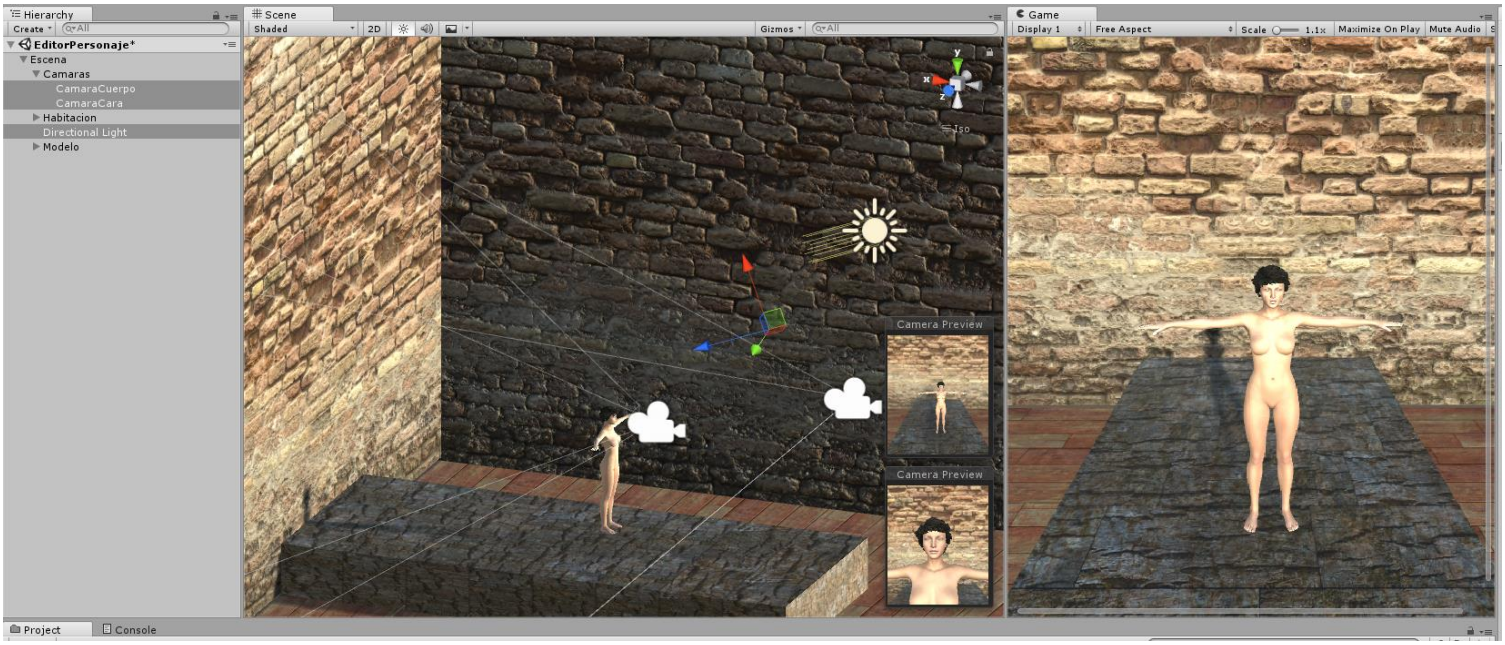

Figura 4.2.1.3: Escena EditarPersonaje

4.2.2 Escena RandomCharacters

En esta escena se generará el número que queramos de modelos distintos a lo largo de un mapa, colocados en posiciones aleatorias y con animaciones.

Los *GameObjects* que componen esta escena son los siguientes:

- **Main Camera:** Es la cámara principal, sigue a nuestro modelo desde atrás allá donde vaya.
- **Modelo:** Es el modelo humanoide que hemos editado.
- **Terreno:** Es el mapa donde se encuentra nuestro modelo. Está compuesto de numerosos *GameObjects* (Árboles, arbustos, un terreno y viento)
- **Directional light:** Determina la iluminación de la habitación y el sombreado deseado de los GameObjects que componen la escena.

Los Scripts que se utilizan en esta escena son los siguientes:

- **CreateProceduralCharacters.cs:** Crea de forma procedural el número de modelos que le indiquemos y los coloca de forma aleatoria por todo el terreno.

```
for (int i = 0; i < numeroCharacters; i++){

    positionX = Random.Range(-40, 40);
    positionZ = Random.Range(-40, 40);
    genderProb = Random.Range(-1, 1);
    clone = Instantiate(model, new Vector3(positionX, 0, positionZ), Quaternion.identity);

    scale = Random.Range(0.9f, 1.6f);
    clone.gameObject.transform.localScale = new Vector3(scale, 1, scale);
    scale = Random.Range(0.5f, 1.3f);
    Debug.Log("Gender probability: " + genderProb);

    if (genderProb >= 0){
        maleGender(clone);
    }
    else{
        femaleGender(clone);
    }
    tamañoCadera(clone);

    /*Modificamos los rasgos faciales*/
    tamañoOjos(clone);
    changeAletaNasal(clone);
    changeNariz(clone);
    changeTabique(clone);
    LongitudLabios(clone);
    TamañoLabios(clone);
    formaBarbilla(clone);
    tamañoMejillas(clone);
    tamañoOrejas(clone);
    tamañoCabeza(clone);

    /*Color de Piel*/
    ColorDePiel(clone);

    /*Modificamos el tamaño de las piernas*/
    tamañoPiernas(clone, scale);

    /*Modificamos el tamaño de los brazos*/
    tamañoBrazos(clone, scale);
```

Figura 4.2.2.1: Parte del código del generador procedural

- **AddAnimationController.cs:** Añade animación a los modelos creados de forma procedural.

- **ThirdPersonUserController.cs:** Este *script*, importado del *standard asset* de *Unity*, dota a nuestro modelo principal de un movimiento en tercera persona.

- **ExitGenPro.cs:** Crea el botón de '*Salir'* que nos lleva a la escena de créditos si lo pulsamos.

En la figura 4.2.2.2 podemos observar cómo está estructurado la escena en *Unity*, a la izquierda podemos observar los *GameObjects*, en el centro-izquierda, la pantalla de edición de la escena, en el centro-derecha como se vería al ejecutarlo y a la derecha del todo podemos ver los scripts añadidos

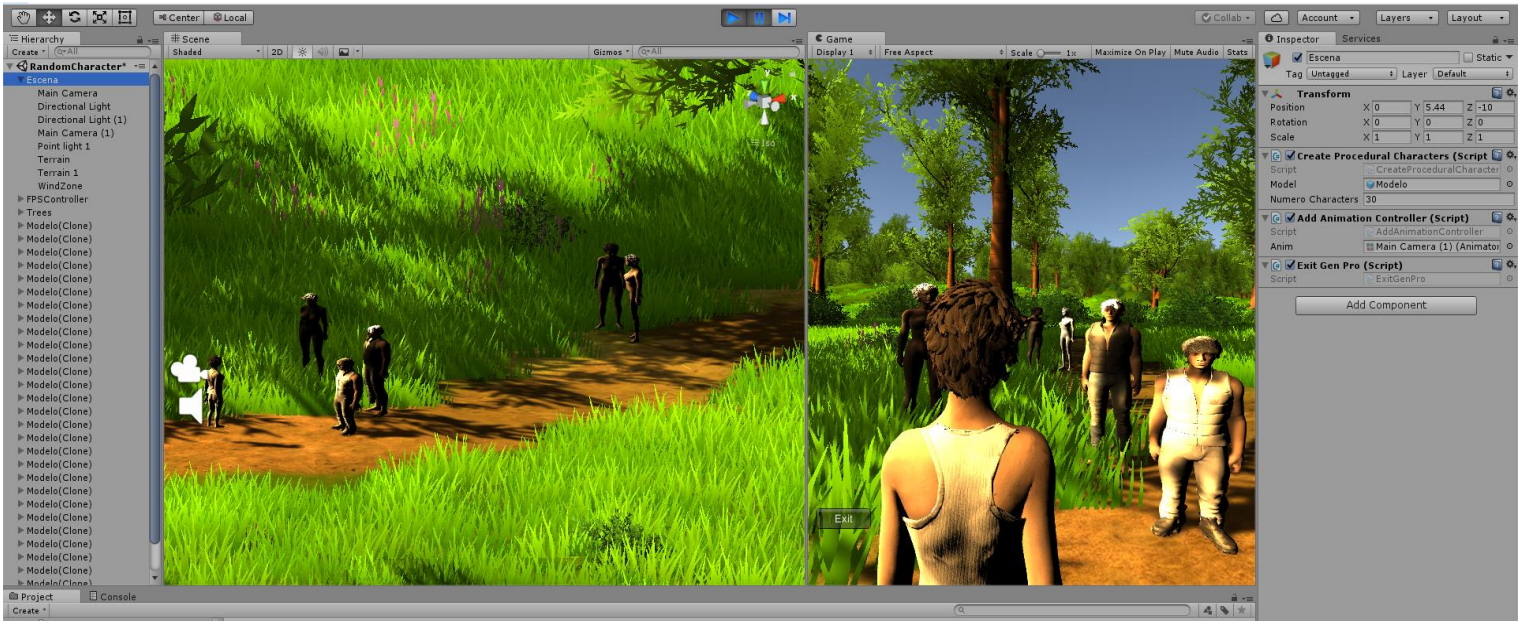

Figura 4.2.2.2: Escena RandomCharacter

Podemos observar la parte de la derecha con mejor detalle en la figura 4.2.2.3, en el s*cript* de *Create Procedural Characters* debemos añadir el modelo que queramos y el número de modelos que queremos generar. Y en *Add Animation Controller* añadimos que animación queremos que tengan esos modelos.

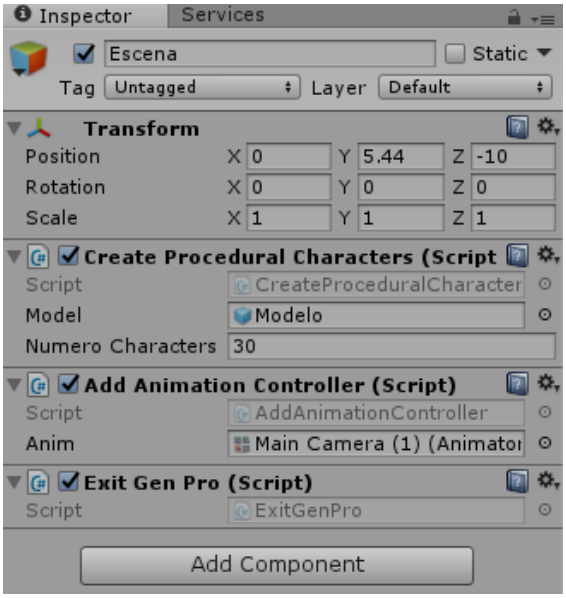

Figura 4.2.2.3: Inspector del GameObject Escena

4.2.3 Escena créditos

Los *GameObjects* que componen esta escena son los siguientes:

- **Main Camera:** Es la cámara principal y única de esta escena. Apunta a un modelo y a los créditos.
- **Modelo:** Es un modelo humanoide que cambiara aleatoriamente cada 10 segundos.
- **Terreno:** Es el mapa donde se encuentra nuestro modelo. Está compuesto de numerosos GameObjects (Árboles, arbustos, un terreno y viento)
- **Creditos:** Son los créditos de la escena.
- **Directional light:** Determina la iluminación de la habitación y el sombreado deseado de los GameObjects que componen la escena.

Los *scripts* que se utilizan en esta escena son los siguientes:

- **RandomCharacters.cs:** Cambia cada 10 segundos las características del modelo que se encuentra en la escena.
- **CloseScene.cs:** Finaliza la escena de créditos.

4.2.4 Escenas de prueba

Las dos escenas que existen en este assets de pruebas son *PruebaOtroModelo* y *PruebaModelosErroneos*. Ambas escenas serán explicadas con mayor detalle en el quinto capítulo de este documento.

4.3 Diagrama de flujo de un script

Este esquema detalla el funcionamiento durante la vida de un *script* en *Unity*.

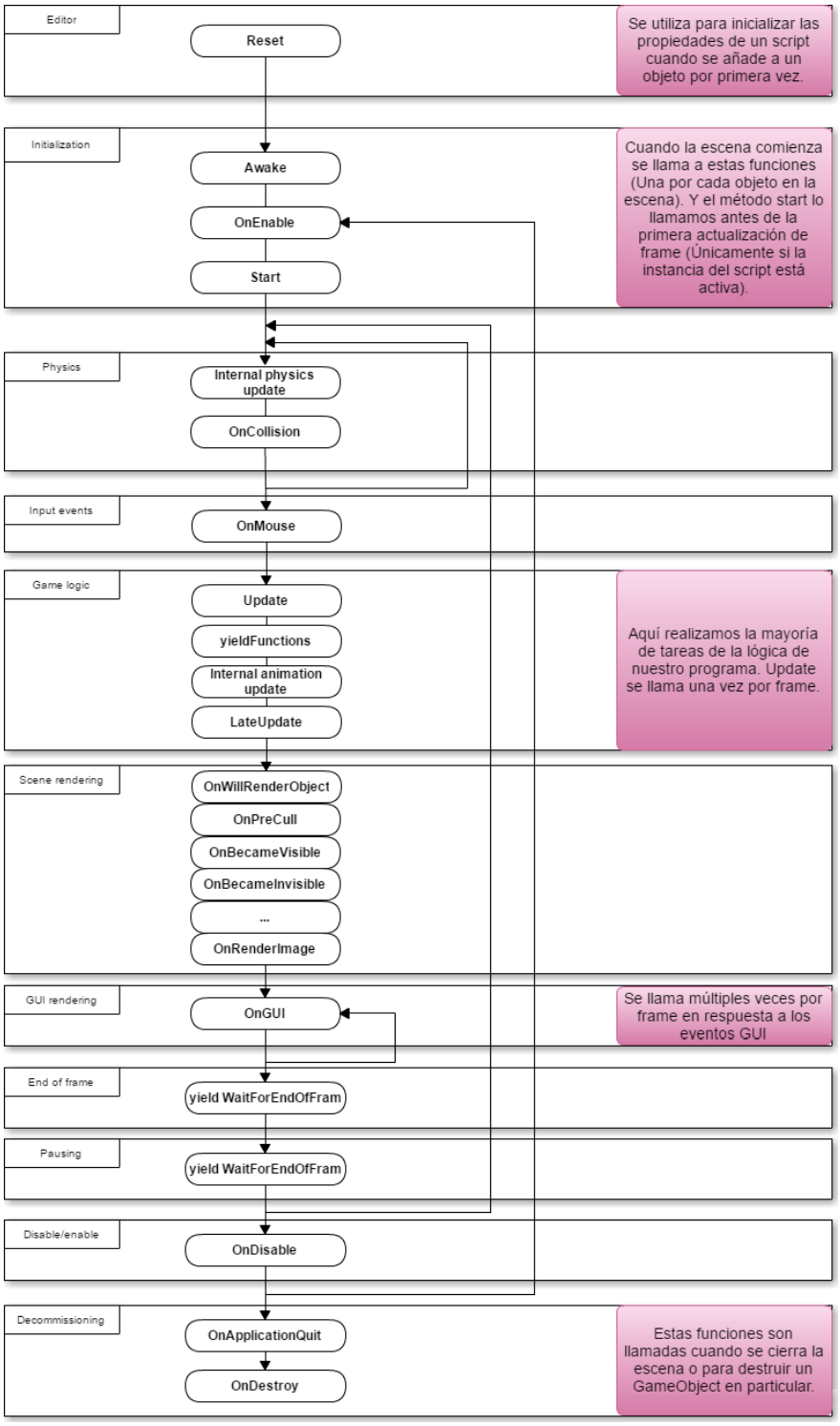

Figura 4.3 Diagrama de flujo de un script

5 Pruebas y resultados

En este capítulo detallaremos las pruebas realizadas del *asset* para validar el correcto funcionamiento del mismo.

5.1 Escenas de pruebas

A continuación, explicaremos las dos escenas realizadas para llevar a cabo las pruebas pertinentes en este proyecto.

5.1.1 Escena *PruebaOtroModelo*

Esta escena sirve para probar la correcta edición de otros modelos que podemos encontrar en la *asset store* o que podemos diseñar nosotros mismos. Es prácticamente la misma escena que la del Editor de modelos, pero más simplificada.

Los *GameObjects* que componen esta escena son los siguientes:

- **CamaraCuerpo:** Es la cámara principal, está activa cuando empieza la escena. Si pulsamos en el botón de edición de cara se desactiva.
- **CamaraCara:** Esta cámara apunta a la cabeza del modelo. Se activa si pulsamos la edición de cara y se desactiva al pulsar edición de cuerpo.
- **Modelo:** Es el modelo humanoide que queremos editar. Este *GameObject* está compuesto por cada uno de los huesos del modelo, que son los hijos de este. Estos *GameObjects* hijos son a los que accederemos con el *script*.
- **Habitacion:** Es la habitación donde se encontrará nuestro modelo al ser editado.
- **Directional light:** Determina la iluminación de la habitación y el sombreado deseado de los *GameObjects* que componen la escena.

Los *scripts* que se utilizan en esta escena son los siguientes:

- **ButtonsController.cs:** Crea todos los botones que hay en la escena (*'Hombre', 'Mujer', 'Aleatorio'*, '*Siguiente, 'Empezar de nuevo' y Salir*) y llama al *script* correspondiente al pulsar cada botón.
- **ChangeCamara.cs:** Aquí tenemos los métodos necesarios para el botón '*Siguiente'.* Cuando pulsamos en el activa y desactiva la cámara correspondiente.
- **FinishEditionTest.cs:** Este script lo utilizamos para finalizar la prueba. Aquí tenemos los métodos necesarios para el botón *'Salir',* al darle clic finalizaremos la prueba.

- **CharacterEditor.cs:** Aquí creamos los métodos para la funcionalidad de los botones *'Hombre', 'Mujer', 'Aleatorio'* y *'Empezar de nuevo'*. Los dos primeros botones modifican el modelo y le cambian la textura según el botón seleccionado. Si pulsamos '*Aleatorio'* nos generara un modelo con unos rangos aleatorios (siempre dentro de unos rangos determinados, evitando que el modelo quede deforme). Y al pulsar en volver a empezar el modelo volverá a los parámetros originales.

  También este *script* nos crea *horizontal sliders* para cada una de las partes del cuerpo a editar (ojos, anchura de la nariz, tamaño de la boca, longitud de las piernas, etc.).

- **HeadEditor.cs:** Contiene los métodos para modificar los rasgos faciales del modelo. Estos métodos serán llamados en CharacterEditor.cs.

- **BodyEditor.cs:** Contiene los métodos para modificar el cuerpo del modelo. Estos métodos serán llamados en CharacterEditor.cs.

Aquí tenemos unas pruebas de los cambios de los rasgos faciales realizadas con el modelo principal que hemos seleccionado:

Figura 5.1.1.1: Cambios rasgos faciales 1

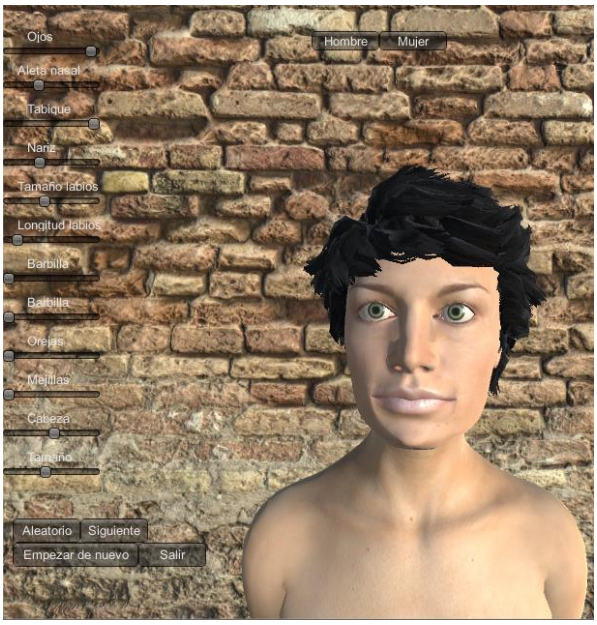

Figura 5.1.1.2: Cambios rasgos faciales 2

Y en las imágenes 5.1.1.3 y 5.1.1.4 vemos los cambios en el cuerpo del modelo:

Figura 5.1.1.3: Cambios Físicos 1

Figura 5.1.1.4: Cambios Físicos 2

Otros modelos humanoides:

Monster:

Este modelo lo podemos encontrar en la *asset store* de manera gratuita. El nombre del asset es *Character Monster 1* y su autor Solum Night.

Figura 5.1.1.5: Modelo Monster 1

Figura 5.1.1.6: Modelo Monster 2

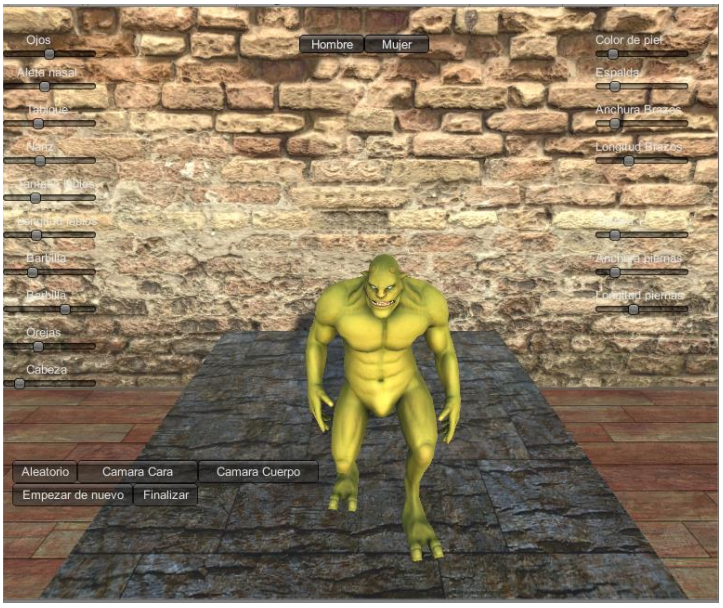

Figura 5.1.1.6: Modelo Monster 3

Figura 5.1.1.7: Modelo Monster 4

Zombie:

Este modelo lo podemos encontrar en la *asset store* de manera gratuita. El nombre del *asset* es *Zombie* y su autor Pxltiger.

Figura 5.1.1.8: Modelo Zombie 1

Figura 5.1.1.9: Modelo Zombie 2

Figura 5.1.1.10: Modelo Zombie 3

Figura 5.1.1.11: Modelo Zombie 4

Dependiendo del modelo y de nuestras preferencias podemos jugar con la creación de los distintos modelos y hacerlos aún más variados. En el caso del *Zombie* tratamos el brazo derecho y el izquierdo por separado pudiendo de esa manera tener un brazo más gordo, largo y fuerte que el otro.

5.1.2 Escena *PruebaModelosErroneos*

En esta escena se generará 10 personajes de un modelo determinado, colocados en fila y con una animación simple para que se muevan en el propio sitio.

Los *GameObjects* que componen esta escena son los siguientes:

- **Main Camera:** Es la cámara principal, sigue a nuestro modelo desde atrás allá donde vaya.

- **WoodenFloor:** Es una superficie plana a la que le hemos añadido una textura de madera.

- **Directional light:** Determina la iluminación de la habitación y el sombreado deseado de los *GameObjects* que componen la escena.

Los *scripts* que se utilizan en esta escena son los siguientes:

- **CreateProceduralTest.cs:** Crea de forma procedural 10 modelos del modelo que le pasemos y los coloca en fila. También se le añade una animación para que se muevan en el propio sitio.

- **FirstPersonUserController.cs:** Este *script*, importado del *standard asset* de *Unity* nos permite manejar un personaje en primera persona y así poder acercanos a los modelos creados.

- **FinishEditionTest.cs:** Crea el botón de '*Salir'* que nos cierra la escena.

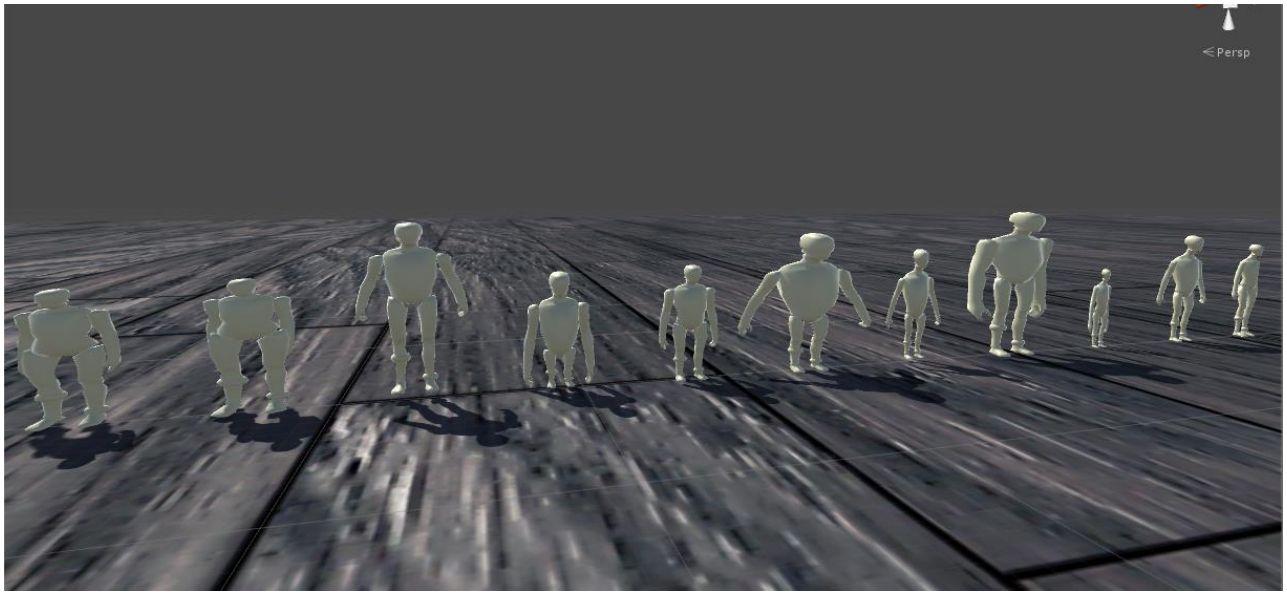

Figura 5.1.2.1: Escena PruebaModelosErroneos 1

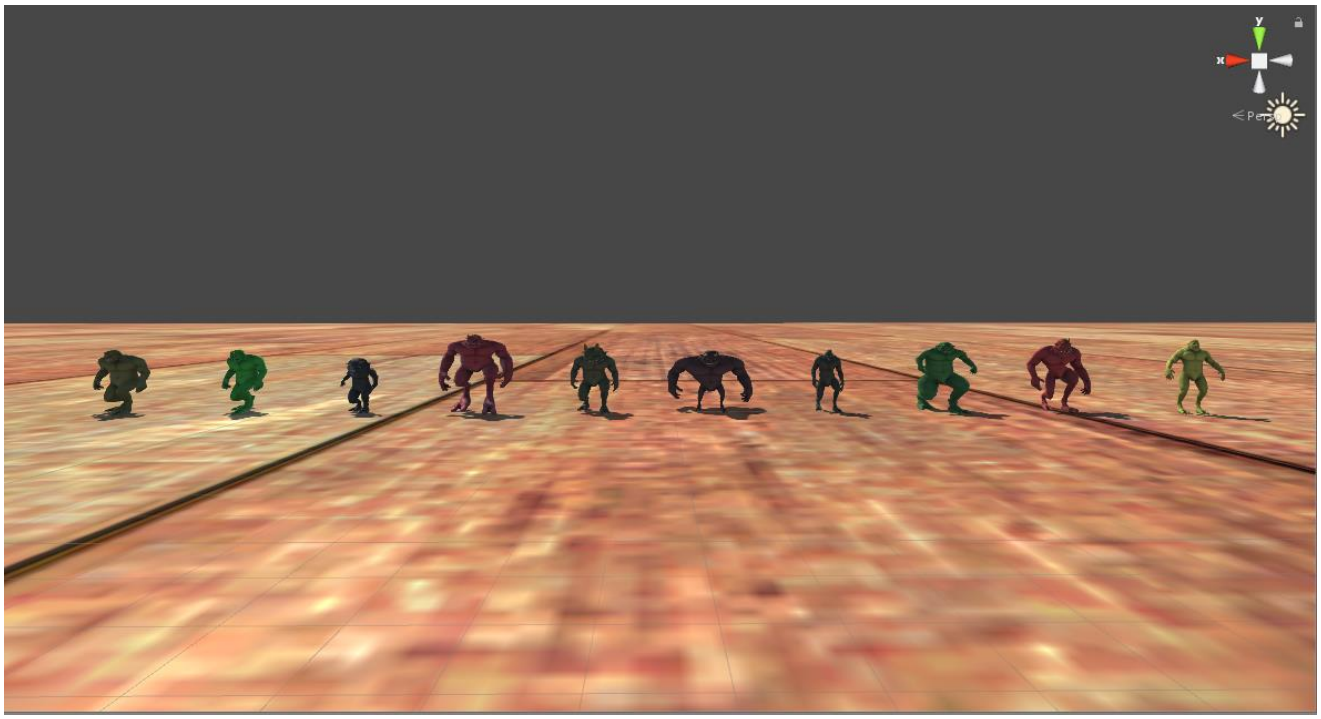

Figura 5.1.2.2: Escena PruebaModelosErroneos 2

Los modelos utilizados en estas pruebas han sido el orco de *Character Monster 1* de *Solum Night*, el modelo humano descargado del *asset Morph3D* y el modelo *Adam* de *Unity Technologies.*

6 Conclusiones y trabajo futuro

6.1 Conclusiones

Las conclusiones de las que hablaremos en este apartado relacionan los objetivos propuestos en esta memoria con los resultados que hemos ido alcanzando durante el desarrollo de este asset para el proyecto.

El objetivo principal de este proyecto consistía en la creación de un asset para Unity que generara de manera procedural modelos humanoides 3D, principalmente para reducir en la medida de lo posible el espacio ocupado por el videojuego o cualquier asset que quiera utilizar este proyecto. Además, gracias a la implementación de este asset ahorraríamos tiempo de desarrollo en cualquier futuro proyecto, al no tener que modelar distintos personajes. También tendríamos una aplicación con personajes muy variados.

Mi intención con el diseño y desarrollo de este *asset* era la generación de personajes humanos realistas de manera algorítmica. Pero el diseño y la implementación son flexibles, de forma que este generador procedural admite múltiples usos, como se explica en el anexo B (Manual del programador) el usuario que se descargue este *asset* podrá modificar los valores que quiera de cada uno de los huesos para crear los modelos a su gusto (Modelos cabezones, modelos con colores extraño, etc.).

Gracias a la realización de las pruebas hemos podido verificar el correcto funcionamiento del *asset*. Así que podemos concluir que el proyecto cumple con los requisitos propuestos al inicio de este documento y, por tanto, con los objetivos de este trabajo de Fin de Grado.

Finalmente destacamos que la generación procedural no es un método reciente, pero con la mejora de las tecnologías y el crecimiento del mundo de los videojuegos (Creación de juegos grandes y variados en el menor tiempo posible) se ha convertido en un método muy popular y potente.

6.2 Trabajo futuro

Se han cumplido los objetivos propuestos dentro del alcance de este proyecto, no obstante, un generador procedural puede tener un número incalculable de mejoras y/o cambios dependiendo del uso que uno le quiera dar. Por ello, el proyecto se ha publicado en la asset store de Unity con la posibilidad de que cualquier persona de la plataforma pueda importar el proyecto y continuar trabajando en él.

Por mi parte, me ha parecido muy interesante trabajar con la tecnología de la generación procedural y me gustaría seguir mejorando y ampliando las funcionalidades del *asset*, así como continuar aprendiendo y desarrollando proyectos relacionados con la generación por procedimientos.

Bibliografía

[1] Paul Merrell and Dinesh Manocha, “Model Synthesis: A General Procedural Modeling Algorithm”, University of North Carolina. http://graphics.stanford.edu/~pmerrell/tvcg.pdf

[2] Israel Fernández, “Procedurally Generated Content: La revolución de los videojuegos es ahora” https://www.xataka.com/

[3] Documentacion de Unity3D. https://docs.unity3d.com

[4] Oliver Franklin-Wallis, “Games of the future will be developed by algorithms, not humans” http://www.wired.co.uk/article/games-developed-by-algorithms

[5] Massive software. http://www.massivesoftware.com/

[6] Noor Shaker, Julian Togelius, and Mark J. Nelson, *“Procedural Content Generation in Games”* http://pcgbook.com/

[7] Phil Wu, “La historia de Elite” http://eliteesp.es/2017/04/09/la-historia-de-elite-i-el-elite-original/

[8] Bitbucket. https://bitbucket.org/

[9] Mechinho.“La apasionante generación por procedimientos” https://gamesbourg.wordpress.com/

[10] Unity. https://unity3d.com/es

[11] Cacoo. https://cacoo.com/

Glosario

GenPro	Nombre de nuestro asset creado.
Asset	Paquetes de Unity que se podrán compartir con los usuarios de la aplicación.
Procedural	Generador de manera algorítmica.
PCG	Generador procedural.

Anexos

A Manual de instalación

1. Para descargarnos e instalar nuestro asset primero debemos irnos a la *asset store* de *Unity*. Primero iremos a la pestaña *Window* y dentro encontraremos la *asset store*. Otra opción será pulsando Ctrl+9 y se nos abrirá también:

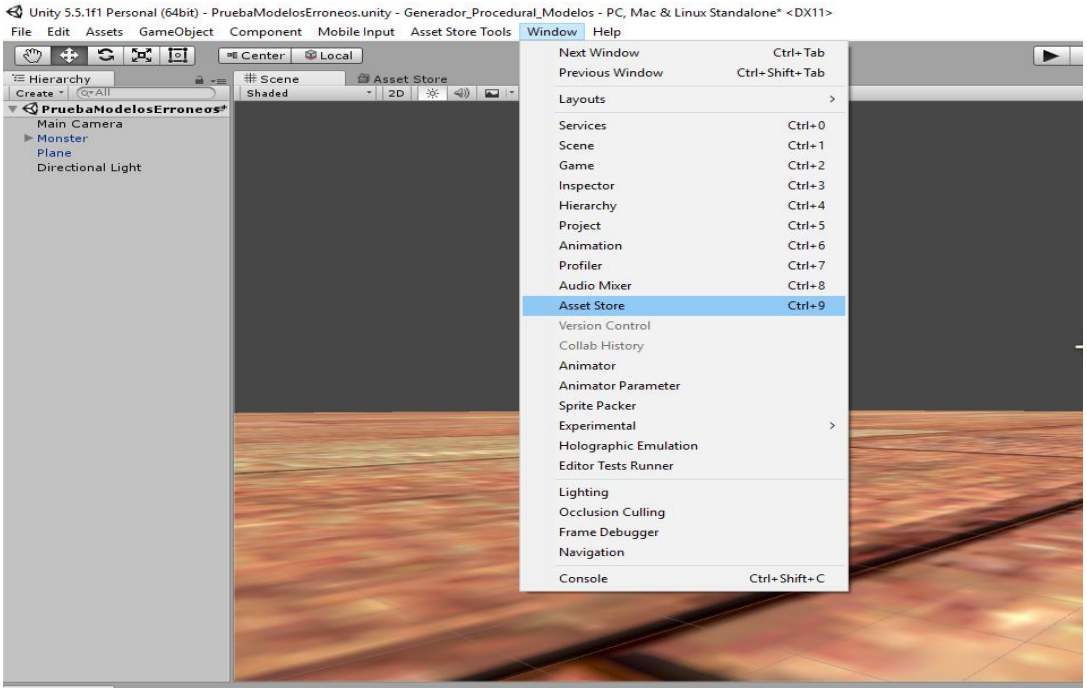

Figura Anexo A: Asset Store

2. Una vez dentro del *asset store* utilizaremos el buscador para descargarnos el *asset* que deseamos (GenPro).

3. Cuando estemos dentro del *asset* GenPro lo único que debemos hacer es pulsar en '*Download'* y esperar a que finalice.

4. Cuando haya finalizado se nos abrirá una nueva ventana donde nos mostrará todo el contenido de nuestro paquete. Hacemos clic en '*Import'* y ya tendremos el *asset* instalado en *Unity*.

B Manual del programador

Uno de mis principales objetivos con la implementación de este *asset* es conseguir que cualquier persona pueda utilizarlo, ampliarlo o modificarlo a su antojo. Como ya hablamos a lo largo de este documento mis intenciones fueron crear modelos humanos realistas; aún así, el *asset* está pensado para que pueda servir para cualquier tipo de modelo humanoide.

Por ejemplo, ahora queremos que nuestro modelo humano tenga cuernos debemos añadirle esos huesos (o usar un modelo humanoide con cuernos) y determinar entre qué tamaño queremos que aparezcan. Una vez hecho esto toca empezar a programar.

Creamos el método TamañoCuernos:

```
public bool TamañoCuernos(GameObject Modelo)
{

    GameObject cuernoIzq = getChildGameObject(Modelo, "lHorn");
    GameObject cuernoDer = getChildGameObject(Modelo, "rHorn");
    float hornScale;
    hornScale = Random.Range (0, 2.25f);

    cuernoIzq.gameObject.transform.localScale = new Vector3(1, hornScale, 1);
    cuernoDer.gameObject.transform.localScale = new Vector3(1, hornScale, 1);

    return true;
}
```

En este caso ambos cuernos siempre tendrán el mismo tamaño, pero eso dependerá de nuestras preferencias, podemos crear también un cuerno más grande que el otro o como queramos (utilizando dos hornScale distintos, cada uno con un valor). Una vez creado este método solo nos queda llamarlo desde el generador procedural y ya generara a nuestro modelo con distinto tamaño de cuernos.